Esercizi Capitolo 12 - Divide-et-Impera

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 Analisi Quicksort (Esercizio 12.4 del libro)

Si analizzi la complessità di QuickSort() nel caso in cui tutti gli elementi del vettore da ordinare abbiano lo stesso valore.

Soluzione: Sezione 2.1

1.2 Merge Sort iterativo (Esercizio 12.5 del libro)

Si fornisca una versione iterativa della procedura MergeSort() senza usare una pila e mantenendone la complessità di $O(n \log n)$.

Soluzione: Sezione 2.2

1.3 Prodotto di numeri complessi (Esercizio 12.8 del libro)

Si individui un algoritmo per moltiplicare due numeri complessi usando solo tre operazioni aritmetiche di moltiplicazione anziché quattro.

Soluzione: Sezione 2.3

1.4 Chi manca? (Esercizio 12.11 del libro)

Sia dato un vettore ordinato $A[1 \ldots n]$ contenente n elementi interi distinti appartenenti all'intervallo $1 \ldots n+1$. Si scriva una procedura, basata sulla ricerca binaria, per individuare in tempo $O(\log n)$ l'unico intero dell'intervallo $1 \ldots n+1$ che non compare in A.

Soluzione: Sezione 2.4

1.5 Punto fisso

Progettare un algoritmo che, preso un vettore ordinato A di n interi distinti, determini se esiste un indice i tale che $A[i] = i$ in tempo $O(\log n)$.

Soluzione: Sezione 2.5

1.6 Ricerca in vettori unimodulari

Un vettore di interi A è detto unimodulare se ha tutti valori distinti ed esiste un indice h tale che $A[1] > A[2] > \ldots > A[h-1] > A[h]$ e $A[h] < A[h+1] < A[h+2] < \ldots < A[n]$, dove n è la dimensione del vettore. Progettare un algoritmo che dato un vettore unimodulare restituisce il valore minimo del vettore. La complessità dell'algoritmo deve essere $O(\log n)$.

Soluzione: Sezione 2.6

1.7 Merge coprocessor

Supponiamo che esista un "coprocessore speciale" in grado di effettuare il merge di due sequenze di due liste ordinate di m elementi utilizzando $O(\sqrt{m})$ passi, e sia specialmerge() la funziona da chiamare per utilizzare tale coprocessore. Descrivere un algoritmo di ordinamento basato su tale funzione ed analizzare il suo tempo di esecuzione.

Soluzione: Sezione 2.7

1.8 Massima somma di sottovettori

Progettare un algoritmo che, preso un vettore A di n interi (positivi o negativi), trovi un sottovettore (una sequenza di elementi consecutivi del vettore) la somma dei cui elementi sia massima.

Suggerimento L'algoritmo banale, di forza bruta, che analizza tutti i possibili casi ha complessità $O(n^3)$. È possibile scendere a $O(n^2)$ notando che ri-utilizzando calcoli parziali, è possibile evitare di ricalcolare da capo la somma di un sotto-vettore. È possibile fare ancora meglio, $O(n \log n)$, con una strategia divide-et-impera. È infine possibile progettare un algoritmo che risolve il problema in tempo $O(n)$.

Soluzione: Sezione 2.8

1.9 Valore maggioritario

Scrivere un funzione boolean che dato un vettore A di dimensione n, ritorni **true** se e solo se esiste un valore maggioritario (ovvero che compare almeno $\lfloor n/2 \rfloor$+1 volte). Calcolarne la complessità.
Suggerimento: è possibile utilizzare algoritmi divide-et-impera noti per ottenere algoritmi che abbiano complessità $O(n \log n)$ o addirittura $O(n)$.

Soluzione: Sezione 2.9

1.10 Poligoni

Rappresentiamo un poligono convesso come un vettore $V[1 \ldots n]$ tale che:

- ciascun elemento del vettore rappresenta un vertice del poligono tramite un coppia di coordinate $(V[i].x, V[i].y)$;
- V[1] è il vertice con coordinata x minima;

- i vertici appaiono in ordine antiorario.

Per semplicità, si può assumere che le coordinate x e y di tutti i vertici siano distinte. Progettare un algoritmo che trovi il punto con coordinata y massima. Discutere correttezza e complessità dell'algoritmo proposto. Suggerimento: un algoritmo $O(n)$ è banale, un algoritmo $O(\log n)$ è più interessante.

Soluzione: Sezione 2.10

1.11 Algorithms-R-us

L'azienda Algorithms-R-us è quotata in borsa. Supponete di avere tabulato nel vettore $A[1 \ldots n]$ la quotazione in borsa dell'azienda lungo un periodo di tempo lungo n giorni, e che $A[1] < A[n]$. E' possibile dimostrare che esiste almeno una coppia di giorni consecutivi i, $i+1$ tale per cui $A[i] < A[i+1]$. Scrivere un algoritmo $O(\log n)$ che restuisca un indice i in cui questo avviene.

Soluzione: Sezione 2.11

1.12 Samarcanda

Nel gioco di Samarcanda[1], ogni giocatore è figlio di una nobile famiglia della Serenissima, il cui compito è di partire da Venezia con una certa dotazione di denari, arrivare nelle ricche città orientali, acquistare le merci preziose al prezzo più conveniente e tornare alla propria città per rivenderle.
Dato un vettore P di n interi in cui $P[i]$ è il prezzo di una certa merce al giorno i, trovare la coppia di giornate (x, y) con $x < y$ per cui risulta massimo il valore $P[y] - P[x]$. Calcolare la complessità e dimostrare la correttezza. È possibile risolvere il problema in $O(n)$.

Soluzione: Sezione 2.12

[1] http://zargosl.free.fr/boardg-IT-I.html#samarcanda

2 Soluzioni

2.1 Analisi Quicksort (Esercizio 12.4 del libro)

Nel caso tutti gli elementi del vettore abbiano lo stesso valore, qualunque scelta del perno porta ad una divisione squilibrata del vettore, con $n-1$ elementi da un lato e 0 dall'altro. Ricadiamo quindi nel caso pessimo, che come abbiamo visto ha complessità $O(n^2)$.

2.2 Merge Sort iterativo (Esercizio 12.5 del libro)

Una versione iterativa di Merge Sort si basa sulla seguente osservazione: è possibile invocare la funzione Merge() sulla prima coppia di elementi del vettore, sulla seconda coppia, sulla terza coppia e così via. A questo punto, il vettore è composto da una serie di coppie ordinate (con eventualmente un elemento singolo in fondo). È possibile invocare la funzione Merge() sulla prima quartina di elementi del vettore, sulla seconda, e così via. Poi su gruppi di 8, poi su gruppi di 16 e così via fino a quando il vettore non è completamente ordinato.
Il codice per risolvere il problema è il seguente.

```
MergeSort(int[] A, int n)
    int s = 1
    while s < n do
        int p = 1
        while p + s ≤ n do
            Merge(A, p, p + s, min(p + 2 · s − 1, n))
            p = p + 2 · s
        s = s · 2
```

La variabile s rappresenta il numero di elementi che si considerano già ordinati; due gruppi di s elementi vanno ordinati tramite Merge(). s viene raddoppiato ad ogni iterazione del ciclo **while** più esterno, fino a quando non raggiunge o supera n; a quel punto il vettore è gia ordinato.
La variabile p indica il punto iniziale dove trovare due gruppi consecutivi di s elementi, in cui ogni gruppo è già ordinato. Se $p + s > n$, non ci sono due gruppi, ma uno solo, quindi non ha senso chiamare Merge() e il ciclo **while** più interno può terminare.

2.3 Prodotto di numeri complessi (Esercizio 12.8 del libro)

Vogliamo moltiplicare due numeri $a+bi$ e $c+di$. Il risultato è $ac+adi+bci-bd = ac-bd+(ad+bc)i$, ottenibile con quattro moltiplicazioni e due somme/sottrazioni.
La seguente soluzione è dovuta a Gauss (famoso fra l'altro per i suoi lavori sui numeri complessi). Dato l'input a, b, c, d, calcoliamo la parte reale $p_r = ac - bd$ e la parte immaginaria $p_i = ad + bc$ nel modo seguente:

$$\begin{aligned} m_1 &= ac \\ m_2 &= bd \\ p_r &= m_1 - m_2 = ac - bd \\ m_3 &= (a+b)(c+d) = ac + ad + bc + bd \\ p_i &= m_3 - m_1 - m_2 = ad + bc \end{aligned}$$

che si può calcolare con tre moltiplicazioni e cinque somme/sottrazioni.

2.4 Chi manca? (Esercizio 12.11 del libro)

Poiché manca un unico valore k, tutti i valori i , $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore; tutti i valori i, $k < i \leq n$ sono memorizzati nella posizione $i-1$-esima. Troviamo quindi il più alto indice i tale per cui $A[i] = i$ e il valore mancante sarà $k = i + 1$.
L'idea è basata sulla ricerca dicotomica; analizzando le posizioni comprese fra i e j (estremi inclusi), calcoliamo $m = \lceil (i + j)/2 \rceil$. Se $A[m] = m$, il più alto indice tale per cui $A[i] = i$ è compreso fra m ed j; se $A[m] > m$, il più alto indice tale per cui $A[i] = i$ è compreso fra i ed $m - 1$. Quando ci si riduce ad un solo elemento ($i = j$), abbiamo trovato il nostro indice.
La chiamata iniziale è missing$(A, 1, n)$; la complessità è chiaramente $O(\log n)$.

```
int missing(int[] A, int i, int j)
if i == j then
    if A[i] == i then
        % Needed if the missing number is n + 1
        return i + 1
    else
        return i
int m = ⌊(i + j)/2⌋
if A[m] == m then
    return missing(A, m + 1, j)
else
    return missing(A, i, m)
```

2.5 Punto fisso

È sufficiente adattare l'algoritmo di ricerca binaria. Siano i e j gli estremi del sottovettore sotto analisi, e sia $m = \lfloor (i + j)/2 \rfloor$. Si distinguono i seguenti tre casi:

- Se $A[m] = m$ restituisci **true**;
- Se $A[m] < m$ si prosegue a destra, ovvero sugli elementi $A[m + 1 \ldots n]$. Infatti, poiché gli elementi sono tutti distinti, $A[m - 1] \leq A[m] - 1 < m - 1$; proseguendo in questo modo, si può dimostrare facilmente che $A[i] < i$ per ogni $i < m$.
- Se $A[m] > m$ si prosegue a sinistra, ovvero sugli elementi $A[1 \ldots m - 1]$. Infatti, poiché gli elementi sono tutti distinti, $A[m + 1] \geq A[m] + 1 > m + 1$. Proseguendo in questo modo, si può dimostrare facilmente che $a[i] > i$ per ogni $i > m$.

La procedura ricorsiva puntofisso() è chiamata inizialmente con puntofisso$(A, 1, n)$. Il costo della procedura è banalmente $O(\log n)$.

```
boolean fixedPoint(int[] A, int i, int j)
if i > j then
    return false
int m = ⌊(i + j)/2⌋
if A[m] == m then
    return true
else if A[m] < m then
    return fixedPoint(A, m + 1, j)
else
    return fixedPoint(A, i, m − 1)
```

2.6 Vettori unimodulari (VUM)

Ancora una volta, utilizziamo un meccanismo di ricerca binaria per risolvere il problema. Il minimo del vettore è sicuramente $A[h]$, perché minore di tutti gli elementi che lo seguono e lo precedono.
Consideriamo come caso base la situazione in cui ci sia un solo elemento. Questo ovviamente è un vettore uni-modulare "degenere" ma accettabile, quindi restituiamo l'unico valore presente.
Nel caso invece gli elementi siano 2 o più, si considerano l'elemento mediano $A[m]$ e il suo successore $A[m+1]$:

- se $A[m] < A[m+1]$, in questi due elementi il vettore è crescente e quindi il minimo si troverà nel sottovettore di sinistra $A[i \dots m]$, non potendo essere a destra dove il vettore continuerà a crescere;
- se $A[m] > A[m+1]$, in questi due elementi il vettore è decrescente e quindi il minimo si troverà nel sottovettore di destra $A[m+1 \dots j]$, non potendo essere a sinistra dove il vettore continuerà a decrescere;
- per definizione di vettore unimodulare, non può essere che $A[m] = A[m+1]$.

Si noti che utilizzando l'intero inferiore per il calcolo di m, $m+1$ è sicuramente un indice minore di j, anche nel caso particolare in cui gli elementi siano solo due.
Si noti che questo algoritmo funziona correttamente anche quando il valore minimo si trova nel primo o nell'ultimo elemento.

```
int vum(int[] A, int i, int j)
  if i == j then
    return A[i]
  int m = ⌊(i + j)/2⌋
  if A[m] < A[m + 1] then
    return vum(A, i, m)
  else
    return vum(A, m + 1, j)
```

Trattandosi di ricerca dicotomica, la complessità computazionale è ovviamente $O(\log n)$.

2.7 Merge coprocessor

L'algoritmo è ovviamente identico a Merge Sort. Non lo ripetiamo qui.
L'analisi è molto semplice. Ad ogni passo ricorsivo, il vettore viene suddiviso in due parti uguali, a cui viene applicato la nostra funzione di Merge Sort, e la chiamata di merge richiede $(O(\sqrt{n})$. La funzione di ricorrenza è quindi:

$$T(n) = 2T(n/2) + O(\sqrt{n})$$

Utilizzando il master theorem, si deve verificare qual è il rapporto fra la funzione $n^{\log_2 2}$ e la funzione $\sqrt{n}$. Poiché $n^{\log_2 2} = \Omega(n^{1/2})$, la soluzione della ricorrenza è $T(n) = O(n)$.

2.8 Massima somma di sottovettori

Innanzitutto, il vettore di input deve contenere valori negativi, altrimenti la risposta è data dal vettore completo.
Un algoritmo banale e poco efficiente per risolvere questo problema consiste nel considerare tutti gli $n(n+1)/2$ sottovettori, calcolare la loro sommatoria e ritornare il valore più alto. Un algoritmo del genere

ha costo $O(n^3)$. Si noti che il massimo viene inizializzato a zero; infatti, se tutti i valori sono negativi un sottovettore vuoto ha somma zero ed è preferibile.

```
int maxsum(int[] A, int n)
    int max = 0                          % Massimo valore trovato
    for i = 1 to n do
        for j = i to n do
            int sum = sum(A, i, j)       % Somma sottovettore
            max = max(sum, max)
    return max
```

```
int sum(int[] A, int i, int j)
    int sum = 0
    for k = i to j do
        sum = sum + A[k]
    return sum
```

Un algoritmo migliore si può ottenere notando che la somma di $A[i \ldots j]$ è uguale alla somma di $A[i \ldots j-1]$ più $A[j]$. L'algoritmo corrispondente ha costo $O(n^2)$.

```
int maxsum(int[] A, int n)
    int max = 0                          % Massimo valore trovato
    for i = 1 to n do
        int sum = 0                      % Somma sottovettore
        for j = i to n do
            sum = sum + A[j]
            if sum > max then
                max = sum
    return max
```

Un approccio basato su divide-et-impera divide il vettore $A[i \ldots j]$ in due parti $A[i \ldots m]$, $A[m+1 \ldots j]$, con $m = \lfloor (i+j)/2 \rfloor$. Ricorsivamente, viene calcolato:

- max_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra $A[i \ldots m]$
- max_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra $A[m+1 \ldots j]$

Inoltre, viene calcolato iterativamente il massimo sottovettore che sta a cavallo fra la parte sinistra e la parte destra, ottenuto calcolando:

- max'_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra e confinanti con la parte destra (ovvero che terminano in $A[m]$);
- max'_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra e confinanti con la parte sinistra (ovvero che iniziano in $A[m+1]$);

A questo punto, il valore finale è dato dal massimo fra max_d, max_s e $max'_d + max'_s$. Come caso base, si considerano vettori costituiti da 0 elementi (il cui valore è 0) o da un elemento (il cui valore è il massimo fra l'elemento stesso e 0, in modo da escludere valori negativi).

La chiamata iniziale è maxsumRic$(A, 1, n)$; il costo computazionale è $O(n \log n)$.

```
int maxsumRic(int[] A, int i, int j)
    int max_s, max_d, max'_s, max'_d, s, m, k
    if i > j then
        return 0
    if i == j then
        return max(0, A[i])
    m = ⌊(i + j)/2⌋
    max_s = maxsumRic(A, i, m)
    max_d = maxsumRic(A, m + 1, j)
    sum = 0
    max'_s = max'_d = 0
    for k = m downto i do
        sum = sum + A[k]
        if sum > max'_s then
            max'_s = sum
    sum = 0
    for k = m + 1 to j do
        sum = sum + A[k]
        if sum > max'_d then
            max'_d = sum
    return max(max_s, max_d, max'_s + max'_d)
```

Si può fare ancora meglio. Sia t_i il valore del massimo sottovettore che termina in i. Se conosco t_{i-1}, posso calcolare t_i osservando due casi: se $A[i] + t_{i-1}$ ha un valore positivo, questo è il valore del massimo sottovettore che termina in i; se invece ha valore negativo, il massimo sottovettore che termina in i ha valore 0. Questo viene calcolato passo passo dalla variabile $here$, che assume il valore max$(here + A[i], 0)$. Poiché il sottovettore di valore massimo deve terminare in qualche posizione, calcolando nella variabile m il valore massimo fra tutti i valori $here$, si ottiene il valore massimo per l'intero vettore. Questo algoritmo ha costo $O(n)$.

```
int maxsum(int[] A, int n)
    int max = 0                          % Massimo valore trovato
    int here = 0                         % Massimo valore che termina nella posizione attuale
    for i = 1 to n do
        here = max(here + A[i], 0)
        max = max(here, max)
    return max
```

2.9 Valore maggioritario

Una possibile soluzione $O(n \log n)$ è basata sull'ordinamento; si ordine prima il vettore che poi viene scandito alla ricerca di ripetizioni.

```
boolean majority(int[]A, int n)
    sort(A, n)                          % Con un algoritmo O(n log n)
    last = A[1]; count = 1
    for i = 2 to n do
        if A[i] == last then
            count = count + 1
            if count > n/2 then
                return true
        else
            last = A[i]
            count = 1
    return false
```

Una possibile soluzione $O(n)$ si basa sulla seguente osservazione: se esiste un elemento maggioritario, questo è anche il valore mediano. Basta quindi calcolare il mediano, utilizzando uno degli algoritmi $O(n)$ visti a lezione, e poi contare quante volte questo valore compare nel vettore. Se compare più di $n/2$ volte, allora abbiamo trovato il nostro valore.

```
boolean majority(int[]A, int n)
    m = median(A, n)                    % Con un algoritmo O(n)
    count = 0
    for i = 1 to n do
        if A[i] == m then
            count = count + 1
    return (count > n/2)
```

2.10 Poligoni

Il fatto che l'algoritmo possa essere eseguito in tempo $O(\log n)$ suggerisce di utilizzare una ricerca dicotomica. Dobbiamo però capire come scegliere il prossimo punto di ricerca (andiamo a destra oppure a sinistra?), e capire quando abbiamo trovato il punto che cerchiamo.
L'idea è questa: dati i punti i e j (inizialmente 1 e n), scegliamo il punto $h = \lfloor (i + j)/2 \rfloor$. Consideriamo i punti $r = (h + 1) \bmod n$ e $l = (h - 1) \bmod n$. Possono darsi quattro casi:

1. $V[h].y > V[l].y, V[h].y > V[r].y$: h è il punto più alto fra l e r, e poichè il poligono è convesso, è anche il punto più alto di tutti. Abbiamo finito.
2. $V[h].y > V[l].y, V[h].y < V[r].y$: la direzione verso l'alto è $l \to h \to r$, quindi scegliamo di andare verso r.
3. $V[h].y < V[l].y, V[h].y > V[r].y$: la direzione verso l'alto è $l \leftarrow h \leftarrow l$, quindi scegliamo di andare verso l.
4. $V[h].y < V[l].y, V[h].y < V[r].y$ è il punto più basso, possiamo andare verso l o verso r indipendentemente.

Se rimangono due punti, basta scegliere quello più basso.

2.11 Algorithms-R-us

La soluzione proposta è basata sulla tecnica divide-et-impera. Generalizzando, vogliamo risolvere il problema di identificare due valori consecutivi crescenti in un vettore $A[i \dots j]$, con $A[i] < A[j]$. Il problema originale corrisponde a $A[1 \dots n]$.
$A[i \dots j]$ può essere ridotto ad **uno** dei sottoproblemi $A[i \dots m]$ e $A[m \dots j]$, dove $m = \lfloor (i+j)/2 \rfloor$ è l'indice mediano fra i e j, in base alle seguenti osservazioni:

- Se $A[i] < A[m]$, allora è possibile considerare il solo sottoproblema $A[i \dots m]$, in cui il primo estremo è minore dell'ultimo.
- Se $A[m] \leq A[i] < A[j]$, allora è possibile considerare il solo sottoproblema $A[m \dots j]$, in cui il primo estremo è minore dell'ultimo.

Il caso base avviene quando siamo rimasti con due soli elementi.
Si noti che applicare ricorsivamente la funzione ad entrambi i sottovettori è inutile e porta ad una complessità di $O(n)$; si noti che è possibile scrivere facilmente questo algoritmo in maniera iterativa.

```
int trova(ITEM [ ] A, int i, int j)
    if i + 1 == j then
        return i
    else
        int m = ⌊(i + j)/2⌋
        if A[i] < A[m] then
            return trova(A, i, m)
        else
            return trova(A, m, j)
```

Chiamata iniziale: $\mathsf{trova}(A, 1, n)$.

2.12 Samarcanda

Si calcoli, per tutti gli indici consecutivi, la differenza (positiva o negativa) e la si registri nel vettore H: $H[i] = P[i+1] - P[i]$. H può essere calcolato in tempo $O(n)$. Si applichi poi l'algoritmo $O(n)$ visto nella Sezione 2.8.

3 Problemi aperti

3.1 Mergesort su lista (Esercizio 12.1 del libro)

Si riscriva la procedura MergeSort() nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendone la stessa complessità $O(n \log n)$.

3.2 Quicksort su lista (Esercizio 12.2 del libro)

Si riscriva la procedura QuickSort() nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendone la stessa complessità media $O(n \log n)$.

3.3 Sottovettori sottili

Dato un vettore V di interi (positivi o negativi) chiamiamo spessore del vettore la differenza tra il massimo e il minimo del vettore. Progettare un algoritmo che, preso un vettore V di n interi ed un intero C, trovi un sottovettore (una sequenza di elementi consecutivi del vettore) di lunghezza massima tra quelli di spessore al più C. La complessità dell'algoritmo deve essere $O(n \log n)$.